CMCS 611-101
Advanced Computer Architecture

Lecture 7

Pipeline Hazards

September 28, 2009

www.csee.umbc.edu/~younis/CMSC611/CMSC611.htm

Lecture's Overview

❑ *Previous Lecture:*

➔ Pipelined hazards

- Pipelining concept is natural
- Start handling of next instruction while current one is in progress

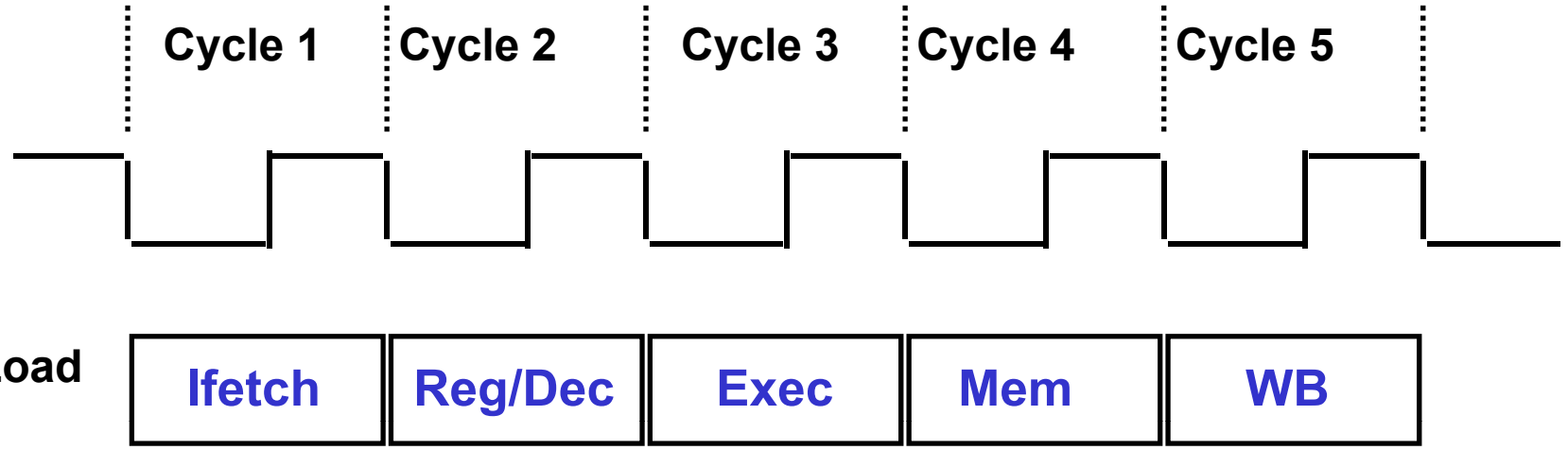
➔ Pipeline performance

- Performance improvement by increasing instruction throughput
- Ideal and upper bound for speedup is number of stages in pipeline

❑ *This Lecture:*

- Structural, data and control hazards
- Data Hazard resolution techniques
- Pipelined control

Stages of Instruction Execution

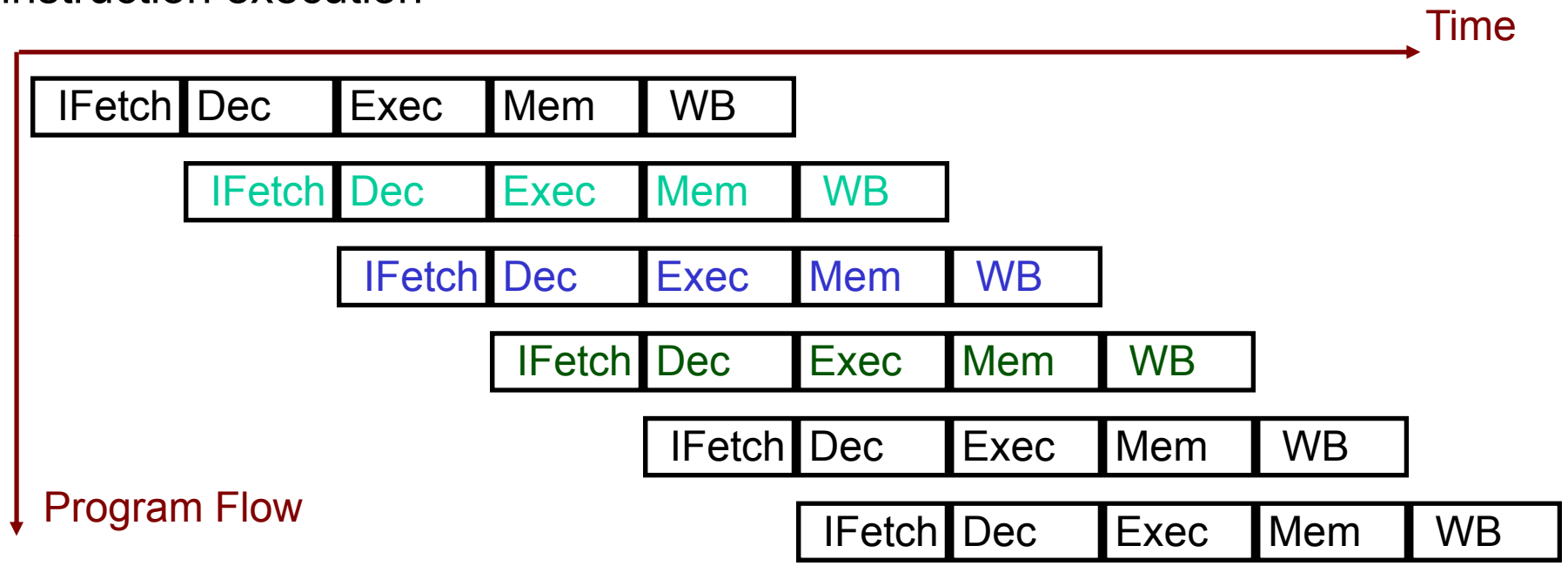


- The load instruction is the longest
- All instructions follows at most the following five steps:
  - ➔ Ifetch: Instruction Fetch
    - Fetch the instruction from the Instruction Memory
  - ➔ Reg/Dec: Registers Fetch and Instruction Decode
  - ➔ Exec: Calculate the memory address
  - ➔ Mem: Read the data from the Data Memory
  - ➔ WB: Write the data back to the register file

* Slide is courtesy of Dave Patterson

Instruction Pipelining

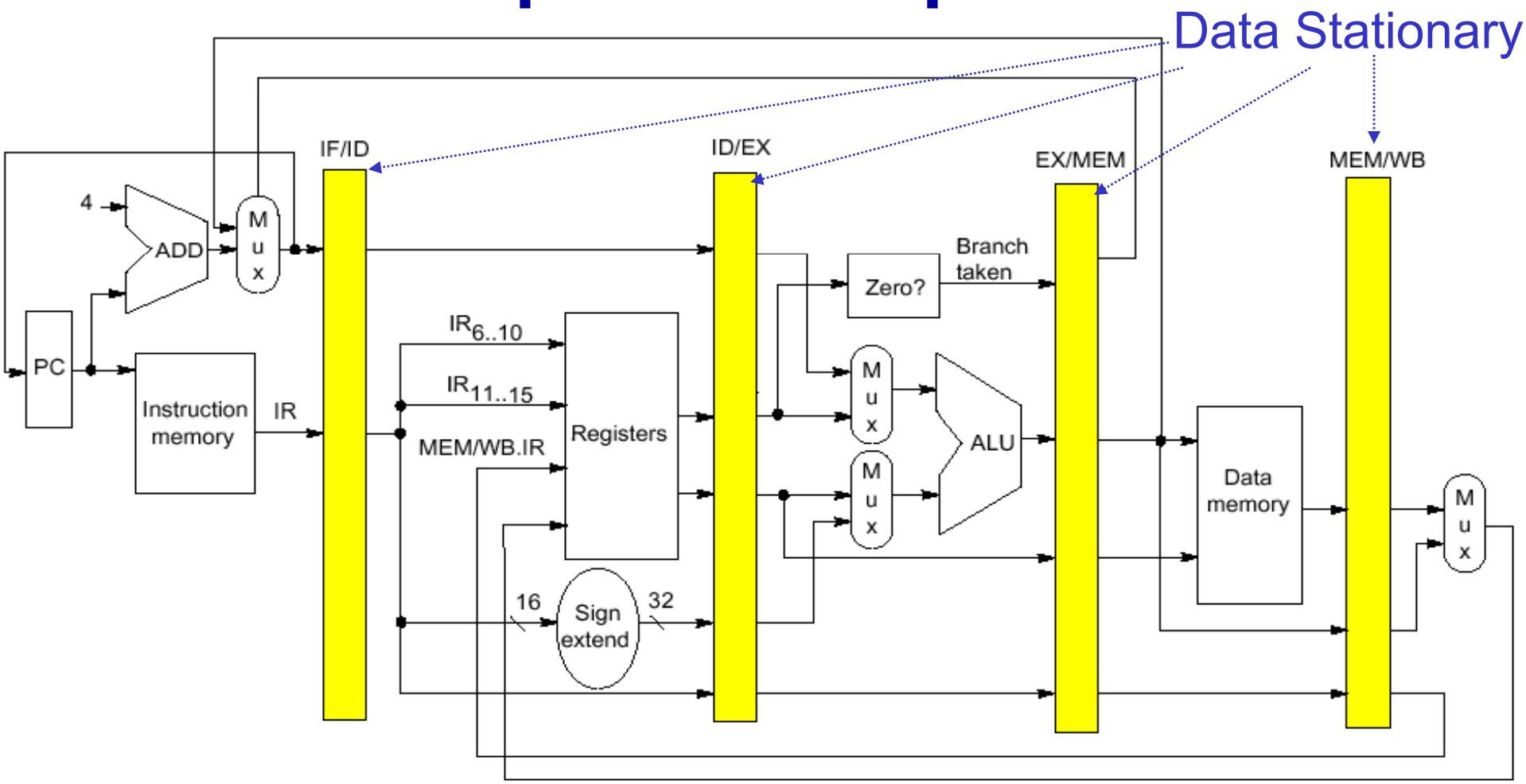
- Start handling of next instruction while the current instruction is in progress
- Pipelining is feasible when different devices are used at different stages of instruction execution



$$\text{Time between instructions}_{pipelined} = \frac{\text{Time between instructions}_{nonpipelined}}{\text{Number of pipe stages}}$$

Pipelining improves performance by increasing instruction throughput

Pipeline Datapath



- Every stage must be completed in one clock cycle to avoid stalls
- Values must be latched to ensure correct execution of instructions
- The PC multiplexer has moved to the IF stage to prevent two instructions from updating the PC simultaneously (in case of branch instruction)

Pipeline Hazards

- Pipeline hazards are cases that affect instruction execution semantics and thus need to be detected and corrected
- Hazards types

Structural hazard: attempt to use a resource two different ways at same time

- ➔ E.g., combined washer/dryer would be a structural hazard or folder busy doing something else (watching TV)
- ➔ Single memory for instruction and data

Data hazard: attempt to use item before it is ready

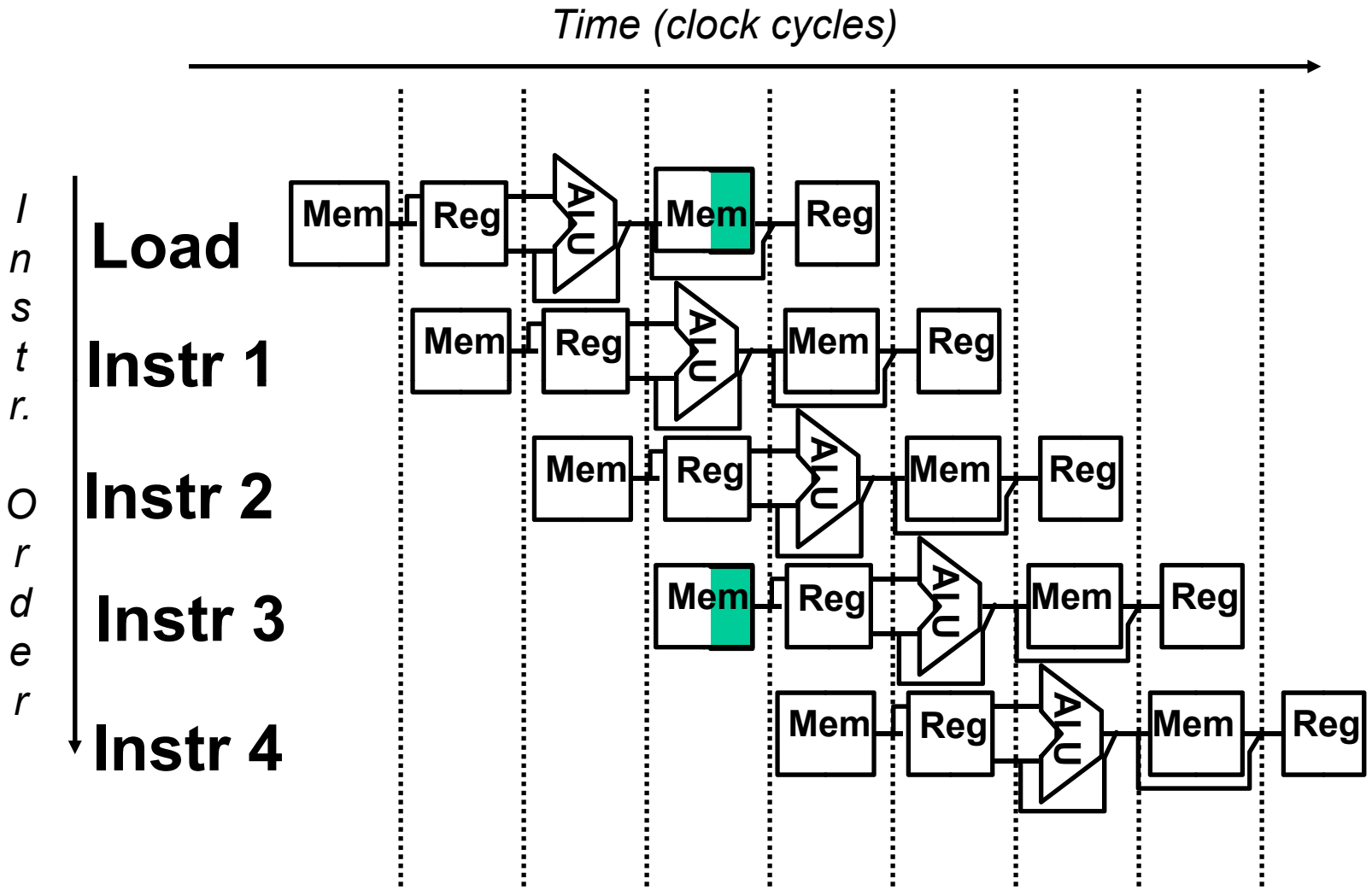
- ➔ E.g., one sock of pair in dryer and one in washer; can't fold until get sock from washer through dryer
- ➔ instruction depends on result of prior instruction still in the pipeline

Control hazard: attempt to make a decision before condition is evaluated

- ➔ E.g., washing football uniforms and need to get proper detergent level; need to see after dryer before next load in
- ➔ branch instructions

- Hazards can always be resolved by waiting

Single Memory is a Structural Hazard



- Can be easily detected
- Resolved by inserting idle cycles

* Slide is courtesy of Dave Patterson

Stalls & Pipeline Performance

$$\text{Speedup from pipelining} = \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}}$$

$$= \frac{\text{CPI unpipelined}}{\text{CPI pipelined}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$

- Ideally the CPI of the pipeline execution is 1 (after fill-up), thus

➔ CPI pipelined = Ideal CPI + Pipeline stall clock per instruction

= 1 + Pipeline stall clock per instruction

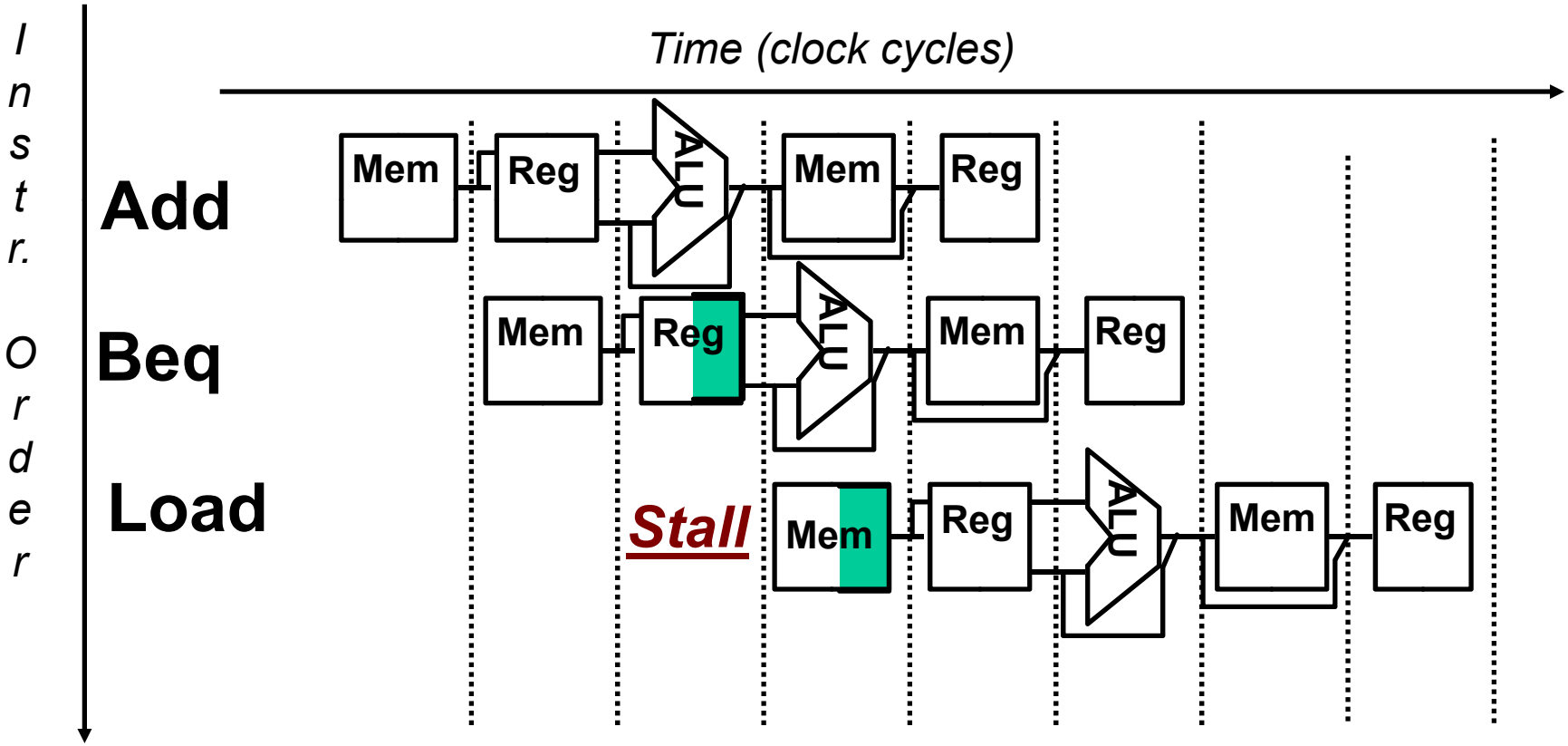
$$\text{Speedup} = \frac{\text{CPI unpipelined}}{1 + \text{Pipeline stall cycles per instruction}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$

- Assuming all pipeline stages are balanced, then

$$\text{Speedup} = \frac{1}{1 + \text{Pipeline stall cycles per instruction}} \times \text{Pipeline depth}$$

Control Hazard

- Stall: wait until decision is clear
  - ➔ It is possible to move up decision to 2nd stage by adding hardware to check registers as being read

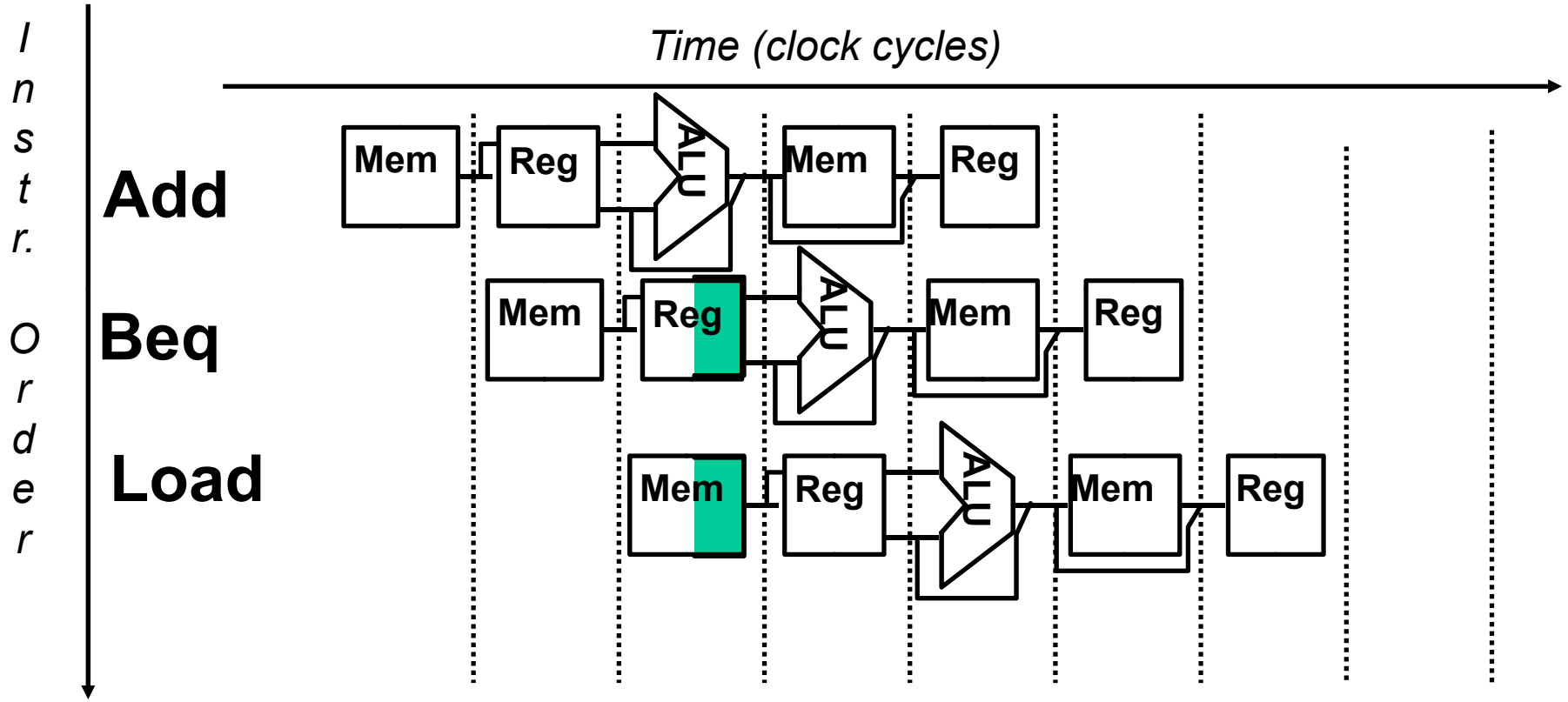


- Impact: 2 clock cycles per branch instruction ⇒ slow

* Slide is courtesy of Dave Patterson

Control Hazard Solution

- Predict: guess one direction then back up if wrong
  - ➔ Predict not taken

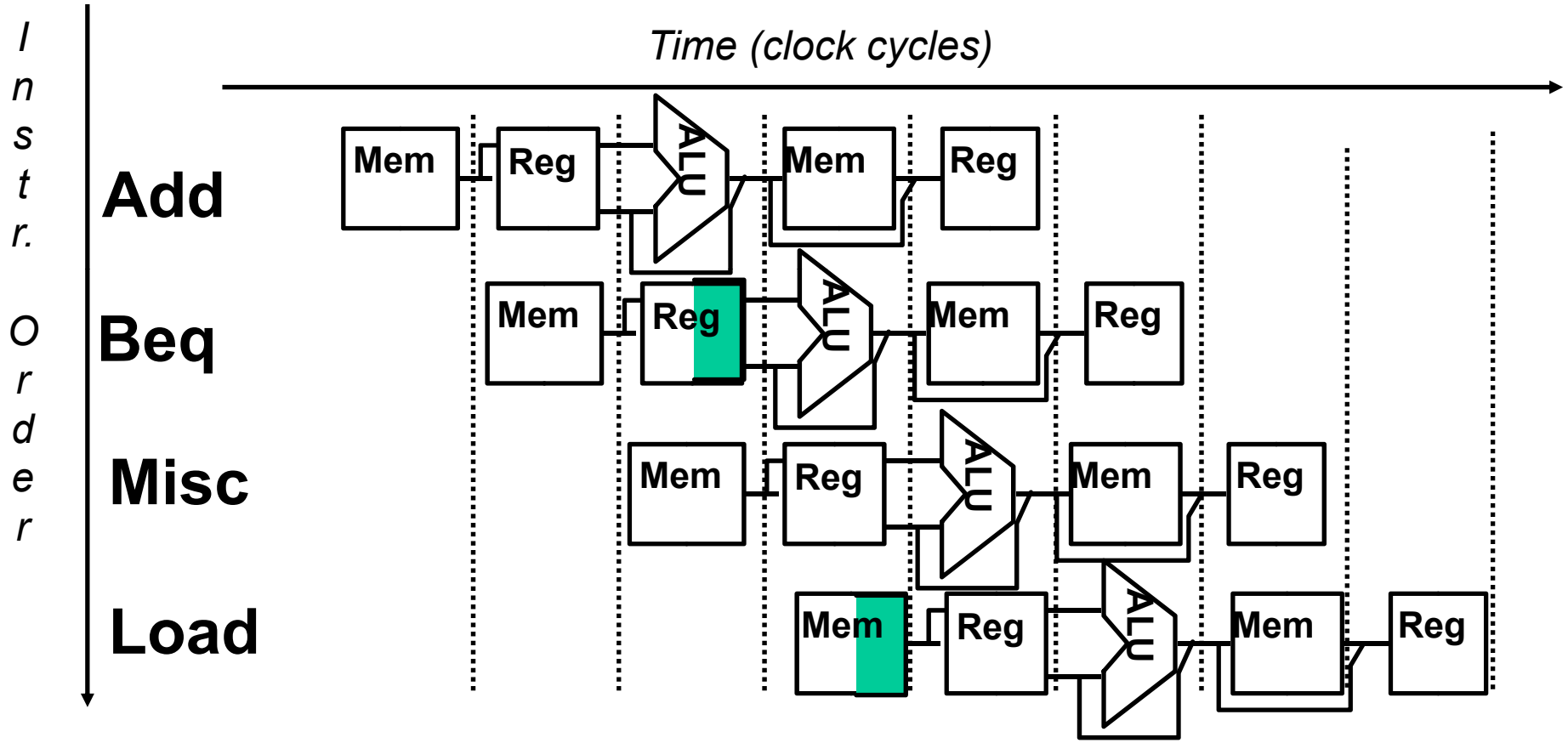


- Impact: 1 clock cycles per branch instruction if right, 2 if wrong (right 50% of time)
- More dynamic scheme: history of 1 branch (90%)

* Slide is courtesy of Dave Patterson

Control Hazard Solution

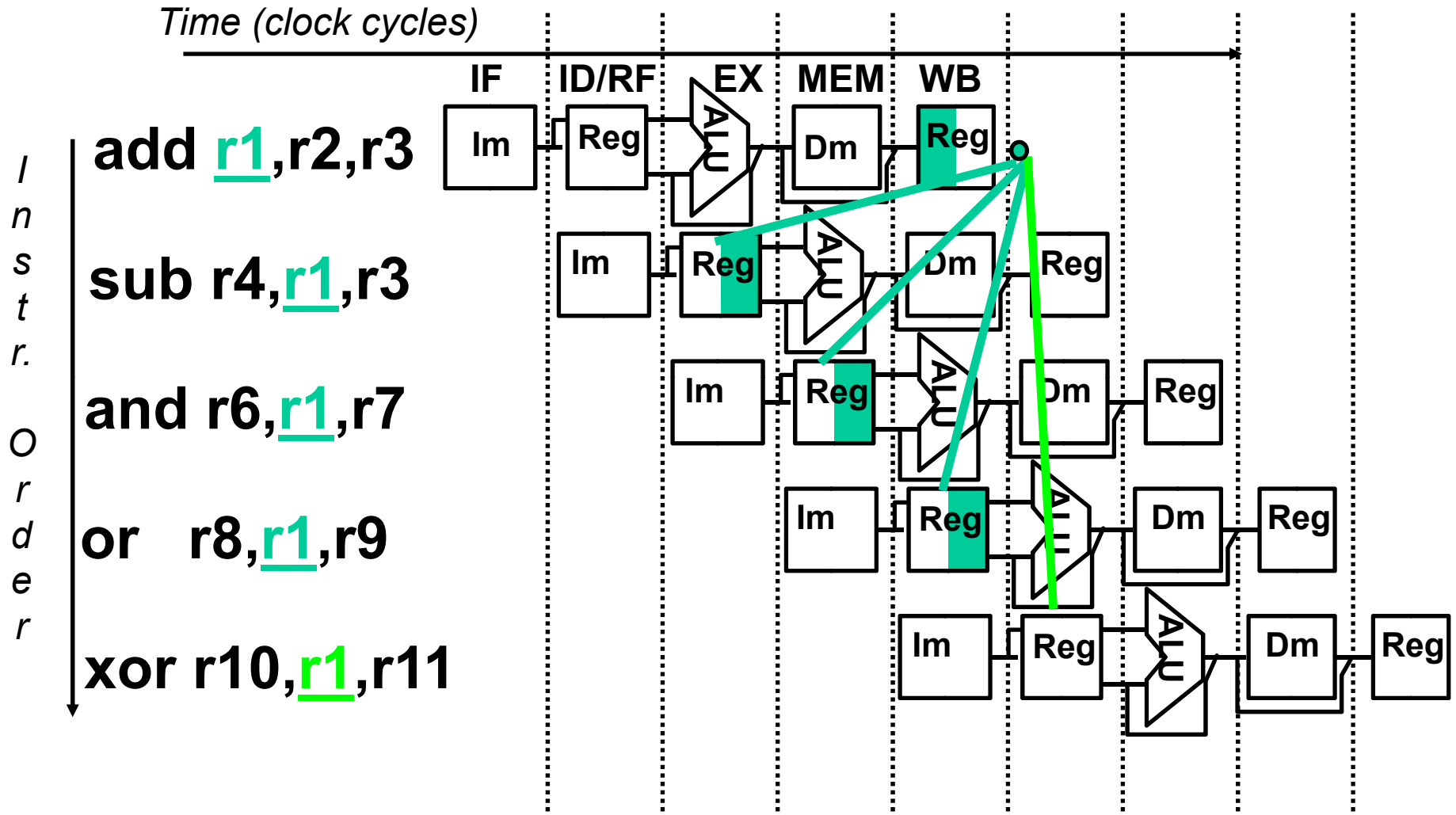
- Redefine branch behavior (takes place after next instruction) "delayed branch"



- Impact: 0 clock cycles per branch instruction if can find instruction to put in "slot" (50% of time)

* Slide is courtesy of Dave Patterson

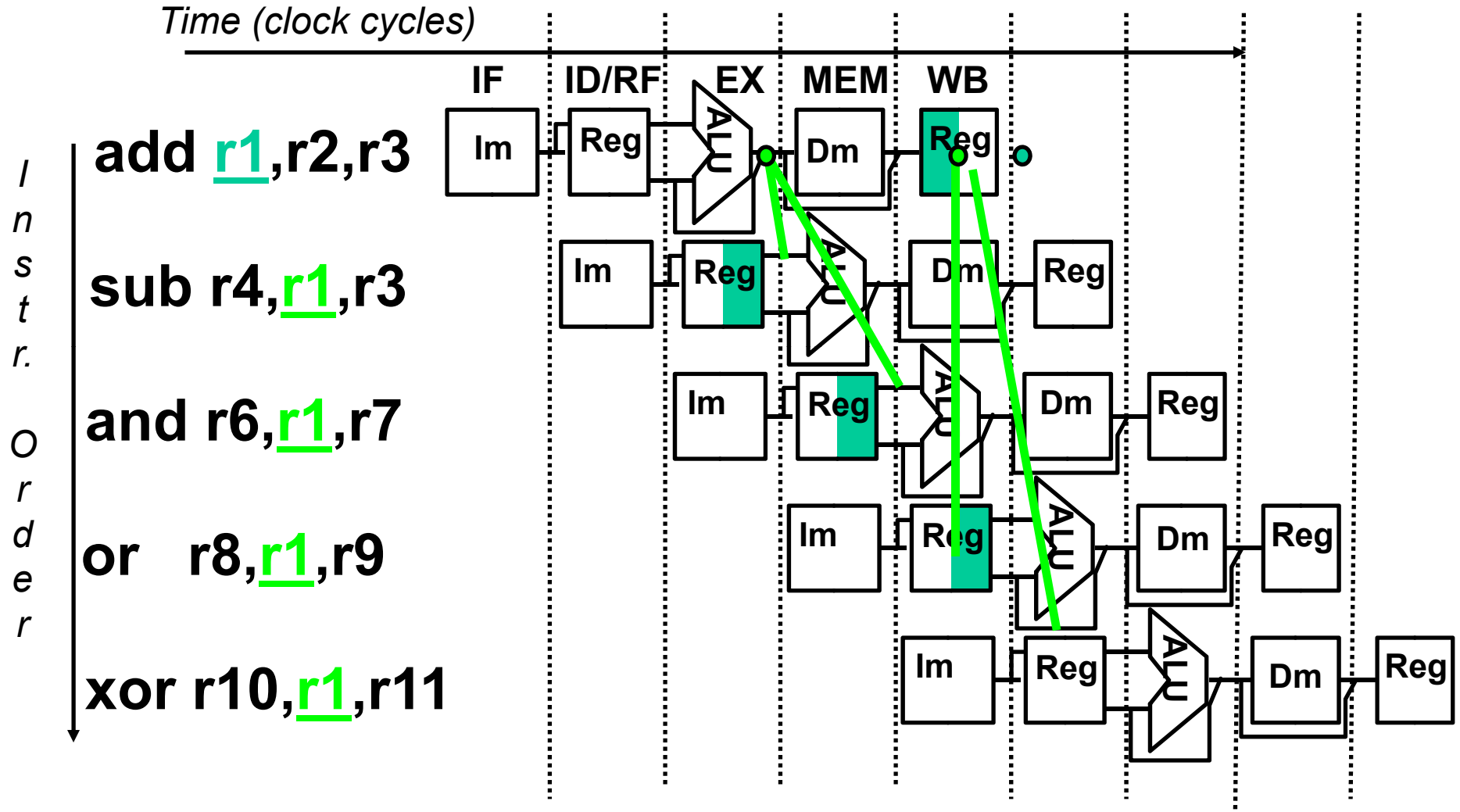
Data Hazard



Dependencies backwards in time are hazards

* Slide is courtesy of Dave Patterson

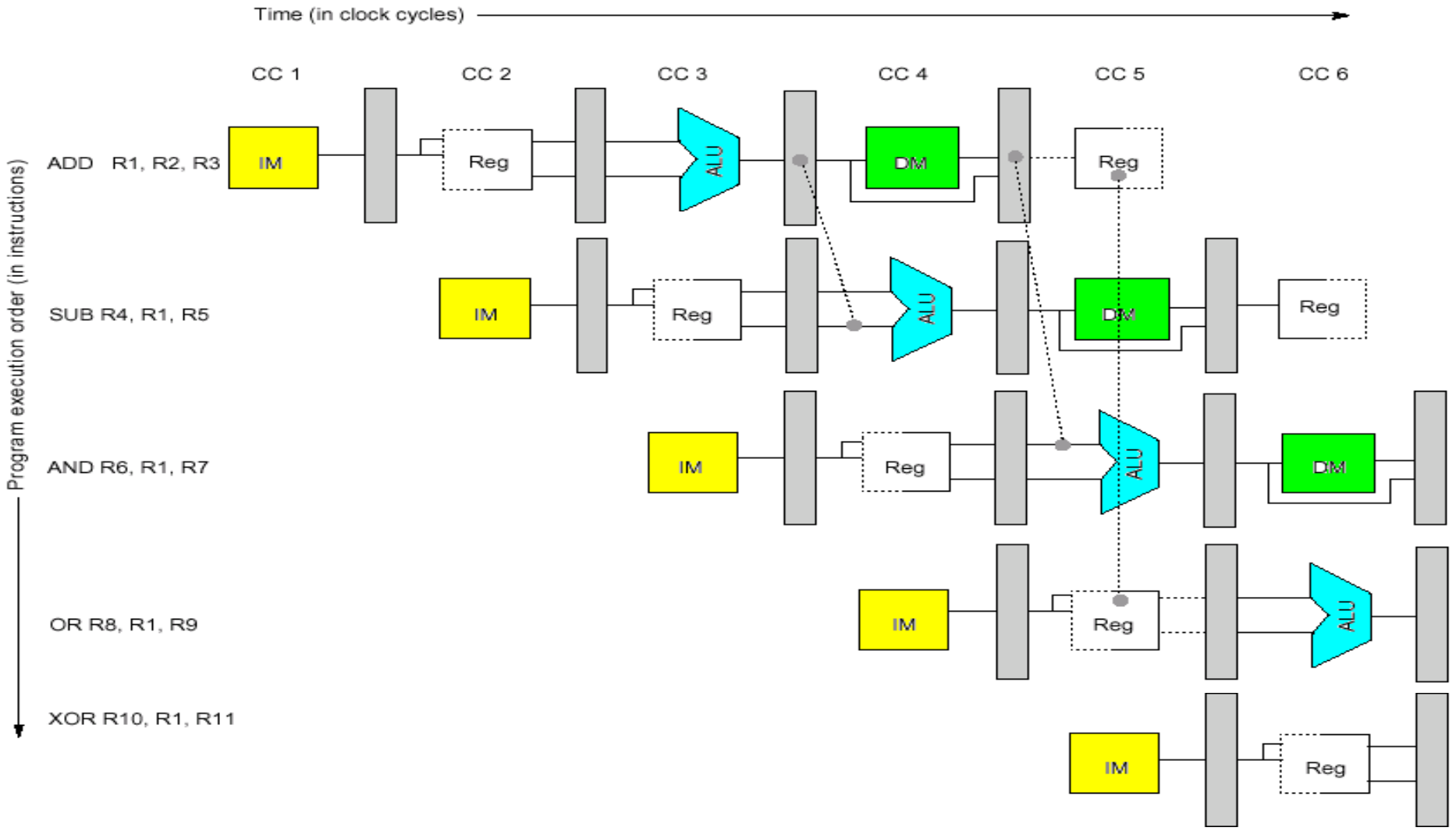
Data Hazard Solution



"Forward" result from one stage to another

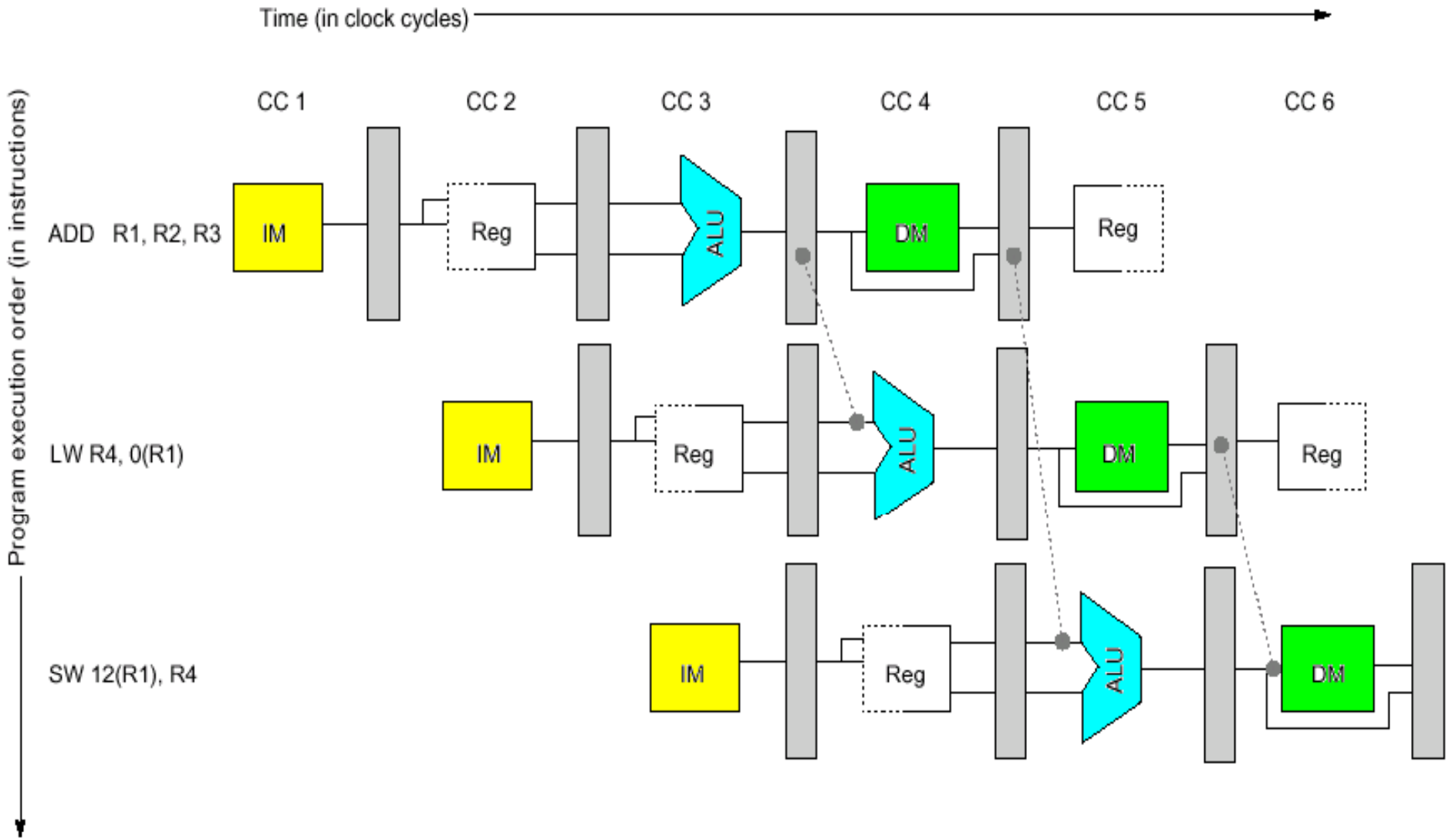
* Slide is courtesy of Dave Patterson

Implementing Data Forwarding



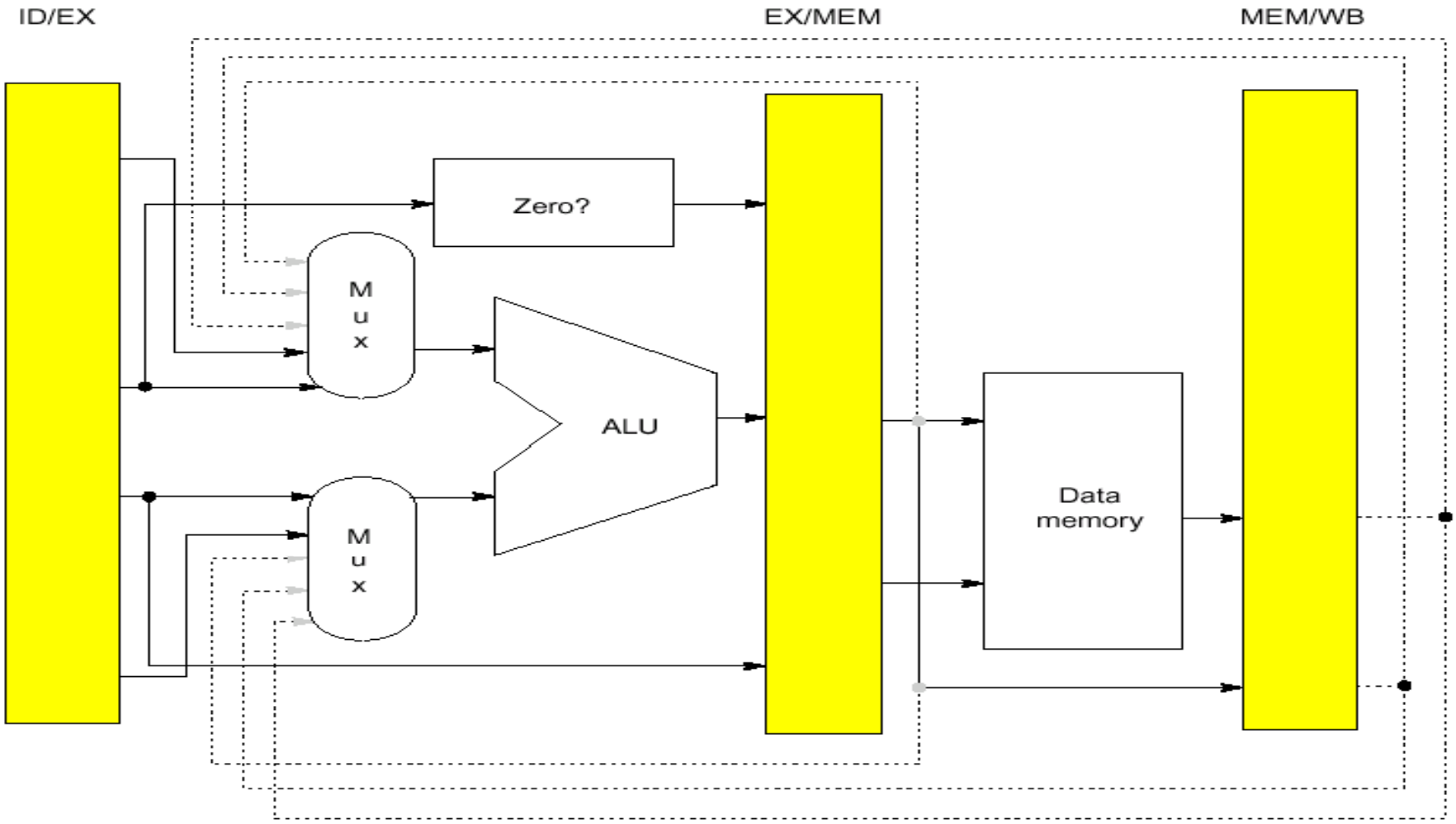
- The ALU result from EX/MEM register is fed back and kept in next stages
- If data hazard is detected the forward values will be used

Example



- The ALU result from EX/MEM register is forwarded to MEM/WB

Forwarding Datapath



- **Three additional inputs are added to the ALU multiplexers each corresponding to a bypass (forwarded data)**

Data Hazards Classification

- Data hazard can happen because dependence among a pair of instructions writing and reading to the same register or memory location
- By stalling the pipeline on cache misses data hazards caused by memory access are avoided
- Data hazards types (instruction *I* proceeds *J*)

RAW (read after write): *J* attempts to read an operand before *I* writes it

➔ Most common type of hazard and typically handled by forwarding

WAW (write after write): *J* attempts to write an operand before *I* writes it

➔ Can happen when writing is done in more than one pipeline stage

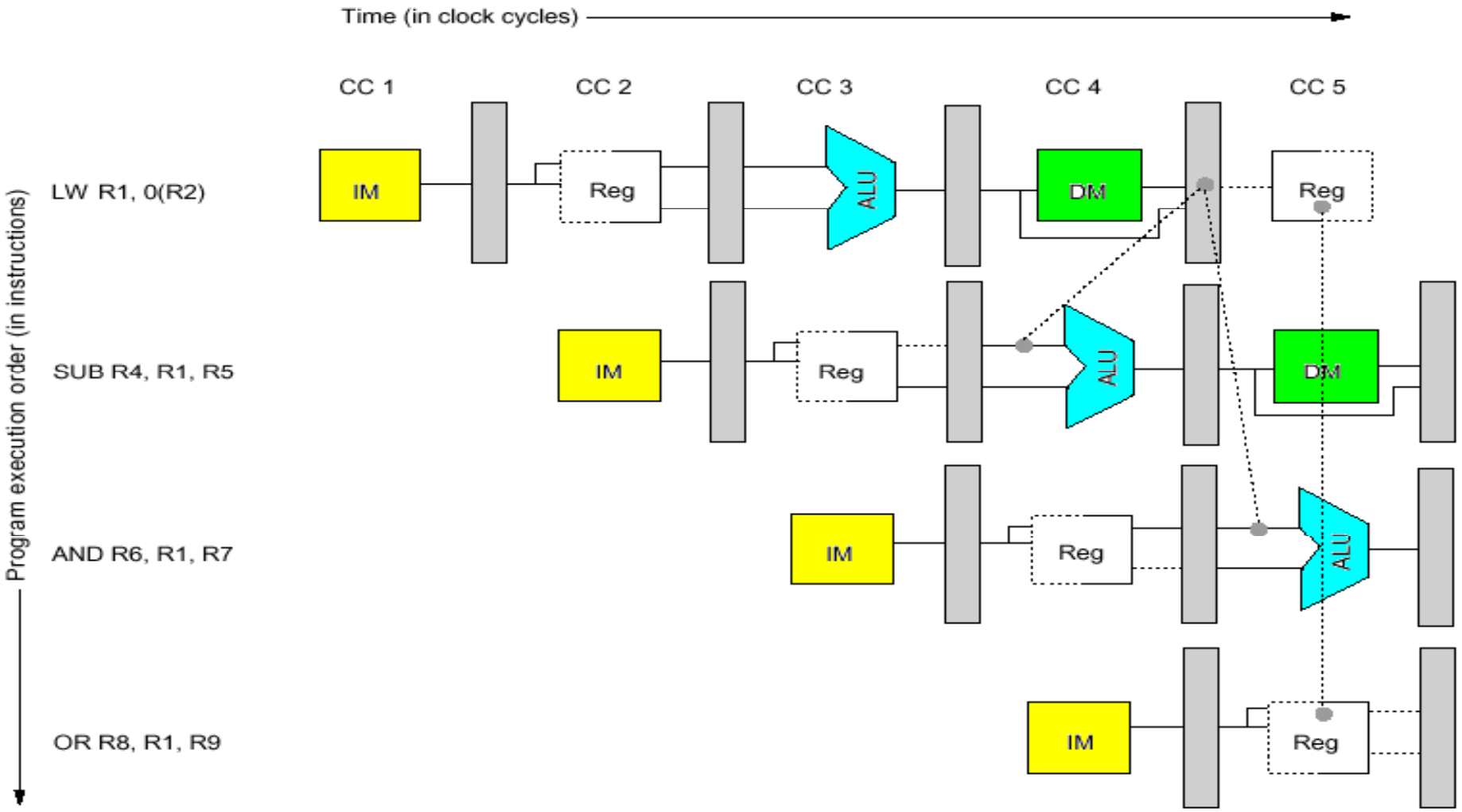
➔ For MIPS pipeline, WAW hazard can happen if WB is performed during MEM stage and the memory is slow so that MEM stage take two cycles

LW R1, 0(R2)	IF	ID	EX	MEM1	MEM2	**WB**
ADD R1, R2, R3		IF	ID	EX	**WB**	

WAR (write after read): *J* attempts to write an operand before *I* reads it

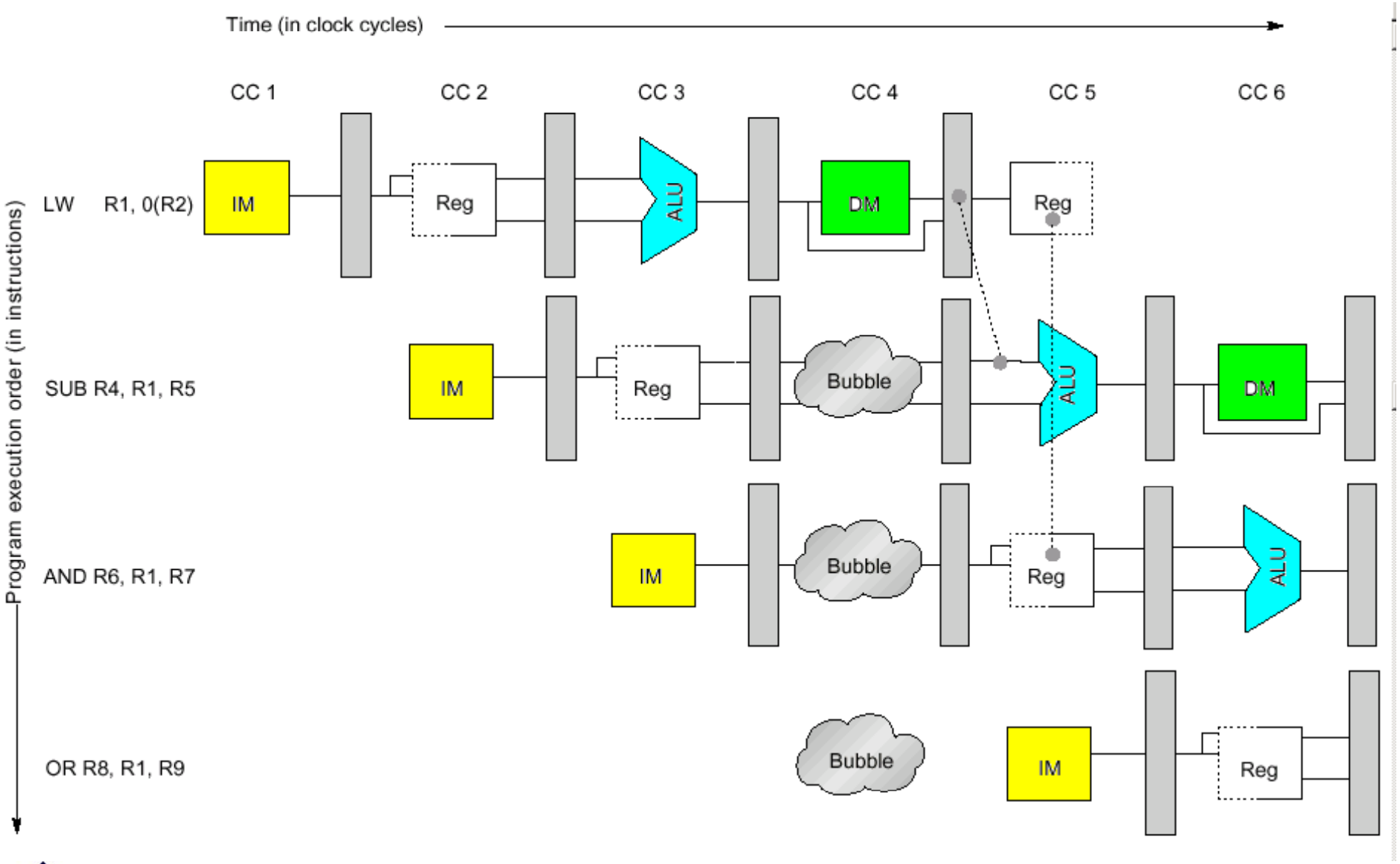
➔ Happen when there are instructions that write early in the pipeline while others reading in a late stage

Data Hazards for Load Instructions



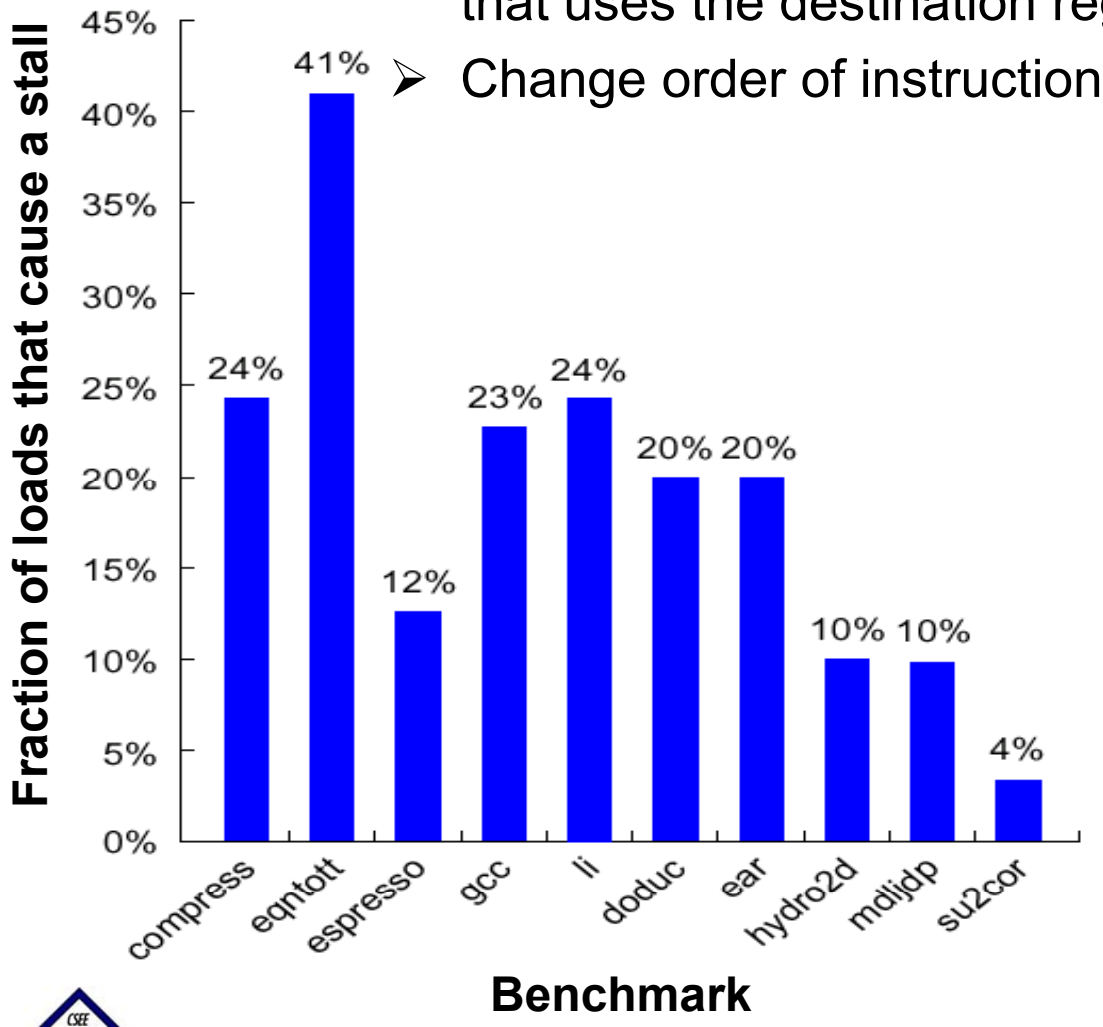
- Dependencies backwards in time but cannot be solved with forwarding
- Must delay/stall instruction dependent on loads

Solving Hazard by Pipeline Interlock



Compiler Scheduling for Data Hazards

- The compiler usually performs *instruction scheduling* to avoid causing data hazard, such as:
  - avoid generating LW followed by an immediate instruction that uses the destination register
  - Change order of instructions in the basic block



Example: compile the following:

a = b + c; d = e – f;

```
LW    Rb, b
LW    Rc, c
LW    Re, e
ADD   Ra, Rb, Rc
LW    Rf, f
SW    a, Ra
SUB   Rd, Re, Rf
SW    d, Rd
```

Swapped

Swapped

Data Hazards Detection

- Detecting hazards early in the pipeline reduces hardware complexity since the machine state will not get erroneously changed
- For the MIPS integer pipeline, all data hazards can be checked in ID stage

Example:

Load interlock detection

Situation	Example code sequence	Action

No dependence	LW **R1**, 45 (R2)
	ADD R5,R6,R7
	SUB R8,R6,R7
	OR R9, R6, R7
		No hazard possible because no dependence exists on R1 in the immediately following three instructions

Dependence requiring stall	LW **R1**, 45 (R2)
	ADD R5,**R1**,R7
	SUB R8,R6,R7
	OR R9, R6, R7
		Comparators detect the use of R1 in the ADD and stall the ADD (and SUB and OR) before the ADD begins EX

Dependence overcome by forwarding	LW **R1**, 45 (R2) ADD R5,R6,R7 SUB R8,**R1** ,R7 OR R9, R6, R7	Comparators detect the use of R1 in the SUB and forward result of load to ALU in time for SUB to begin EX

Dependence with accesses in order	LW **R1**, 45 (R2)
	ADD R5,R6,R7
	SUB R8,R6,R7
	OR R9, **R1**, R7
		No action required because the read of R1 by OR occurs in the second half of the ID phase, while the write of the loaded data occurred in the first half.

Load Interlock Detection

- Pipeline stall is needed when a load instruction is followed by the an instruction that read the yet-to-be-loaded register
- The load interlock conditions for RAW hazards are:

Opcode field of ID/EX ($ID/EX.IR_{0..5}$)	Opcode field of IF/ID ($IF/ID.IR_{0..5}$)	Matching operand fields
Load	Register-register ALU	$ID/EX.IR_{11..15} == IF/ID.IR_{6..10}$
Load	Register-register ALU	$ID/EX.IR_{11..15} == IF/ID.IR_{11..15}$
Load	Load, store, ALU imm., or branch	$ID/EX.IR_{11..15} == IF/ID.IR_{6..10}$

- Control logic is simple combinational circuit with input from ID/EX and IF/ID
- Once the hazard is detected the control unit must insert the pipeline stall and prevent the instructions in the IF and ID stages from advancing
- Since all control logic is derived from the data stationary, stalling the pipeline is simply by setting the ID/EX portion to zero (matching the NOP instruction)
- In case of a stall, the contents of the IF/ID registers will be re-circulated to hold the stalled instruction

Data Forwarding Logic

Pipeline register containing source instruction	Opcode of source instruction	Pipeline register containing destination instruction	Opcode of destination instruction	Destination of the forwarded result	Comparison (if equal then forward)
EX/MEM	Register-register ALU	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	EX/MEM.IR $_{16..20}$ == ID/EX.IR $_{6..10}$
EX/MEM	Register-register ALU	ID/EX	Register-register ALU	Bottom ALU input	EX/MEM.IR $_{16..20}$ == ID/EX.IR $_{11..15}$
MEM/WB	Register-register ALU	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	EX/MEM.IR $_{16..20}$ == ID/EX.IR $_{6..10}$
MEM/WB	Register-register ALU	ID/EX	Register-register ALU	Bottom ALU input	EX/MEM.IR $_{16..20}$ == ID/EX.IR $_{11..15}$
EX/MEM	ALU immediate	ID/EX	Register-register ALU, ALU immediate load, store, branch	Top ALU input	EX/MEM.IR $_{16..20}$ == ID/EX.IR $_{6..10}$
EX/MEM	ALU immediate	ID/EX	Register-register ALU	Bottom ALU input	EX/MEM.IR $_{16..20}$ == ID/EX.IR $_{11..15}$
MEM/WB	ALU immediate	ID/EX	Register-register ALU, ALU immediate load, store, branch	Top ALU input	EX/MEM.IR $_{16..20}$ == ID/EX.IR $_{6..10}$
MEM/WB	ALU immediate	ID/EX	Register-register ALU	Bottom ALU input	EX/MEM.IR $_{16..20}$ == ID/EX.IR $_{11..15}$
MEM/WB	Load	ID/EX	Register-register ALU, ALU immediate load, store, branch	Top ALU input	EX/MEM.IR $_{16..20}$ == ID/EX.IR $_{6..10}$
MEM/WB	load	ID/EX	Register-register ALU	Bottom ALU input	EX/MEM.IR $_{16..20}$ == ID/EX.IR $_{11..15}$

Conclusion

❑ *Summary*

➔ Pipeline Hazards

- Structural, data and control hazards

➔ Data Hazards

- Forwarding techniques for simple data hazards resolution
- Data hazards classifications and detection logic
- Load-caused pipeline stalls and how to limit their scope
- Compiler-based instruction scheduling to avoid pipeline stalls
- Implementation of data hazard detection and forwarding logic

❑ *Next Lecture*

➔ Pipeline control hazards

➔ Pipelining and exception handling

Reading assignment includes Appendix A.2 & A.3 in the textbook